MarXbot User Manual
Version 1.1

Philippe Rétornaz, Stéphane Magnenat, Florian Vaussard
Mobots group - LSRO - EPFL

December 12, 2014

Contents

Foreword

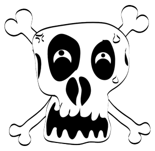
This user manual is devoted to document the marXbot basic use with the bootloaders and ASEBA virtual machines already programmed. This is not a documentation about the electronics nor about embedded C programming on the microcontrollers.

As the marXbot is running an embedded Linux system, most of the tricks in this manual are given for a Linux host system, apart ASEBA which is a cross-platform software. When known to work, the Windows equivalence is given. If you find some working tools, feel free to share with us for future users. Anyway, a Linux system is strongly encouraged.

You will see the following warnings along the road. Please, always read them and make sure you understand them. Feel free to ask us any questions you may have, as some operations could easily break the marXbot.

Mind this point.

This can be harmful for you or the robot. Take great care.



This can destroy the robot. Do this at your own risks.

This can destroy the robot and bring you years of torments, maybe for eternity... You have been warned.

Chapter 1

The MarXbot Robot

This chapter will give you a first crash-course to the marXbot robot, if you are not already familiar with it. At the end, you should be able to understand the basic architecture, both from the hardware and software point of view, and establish a first SSH connection with the robot. For a more in-depth description, please refer to [1–3].

1.1 Overview

An overview of the robot can be found in Fig. 1.1. The marXbot is a general purpose, all-terrain, modular robot. It has been designed and manufactured by the Mobots group[1], belonging to the EPFL – LSRO[2] robotics laboratory.

The robot is modular by design:

1. The base module. It is made up of two wheels (with tracks), the power electronics and a great number of sensors: 24 IR proximity rangers, IR ground rangers (8 long range + 4 short range), 3D accelerometer, 3D gyroscope, RFID reader - writer. The power electronics is capable to hot-swap the battery, allowing the user to exchange the battery without shutting down the power. The main processor will continue to work, while the less useful parts (motors, sensors,...) will be powered off to save energy. There should be enough energy to power the processor for 10 seconds.

   The hot-swap is achieved using supercapacitors. Thus, a minimum charging time of **5 minutes** should be granted before trying this capability.

2. (Optional) Attachment module. This module allows marXbots to attach to each other in swam experiments. The gripper can rotate around the robot. This module also provides 12 RGB LEDs.

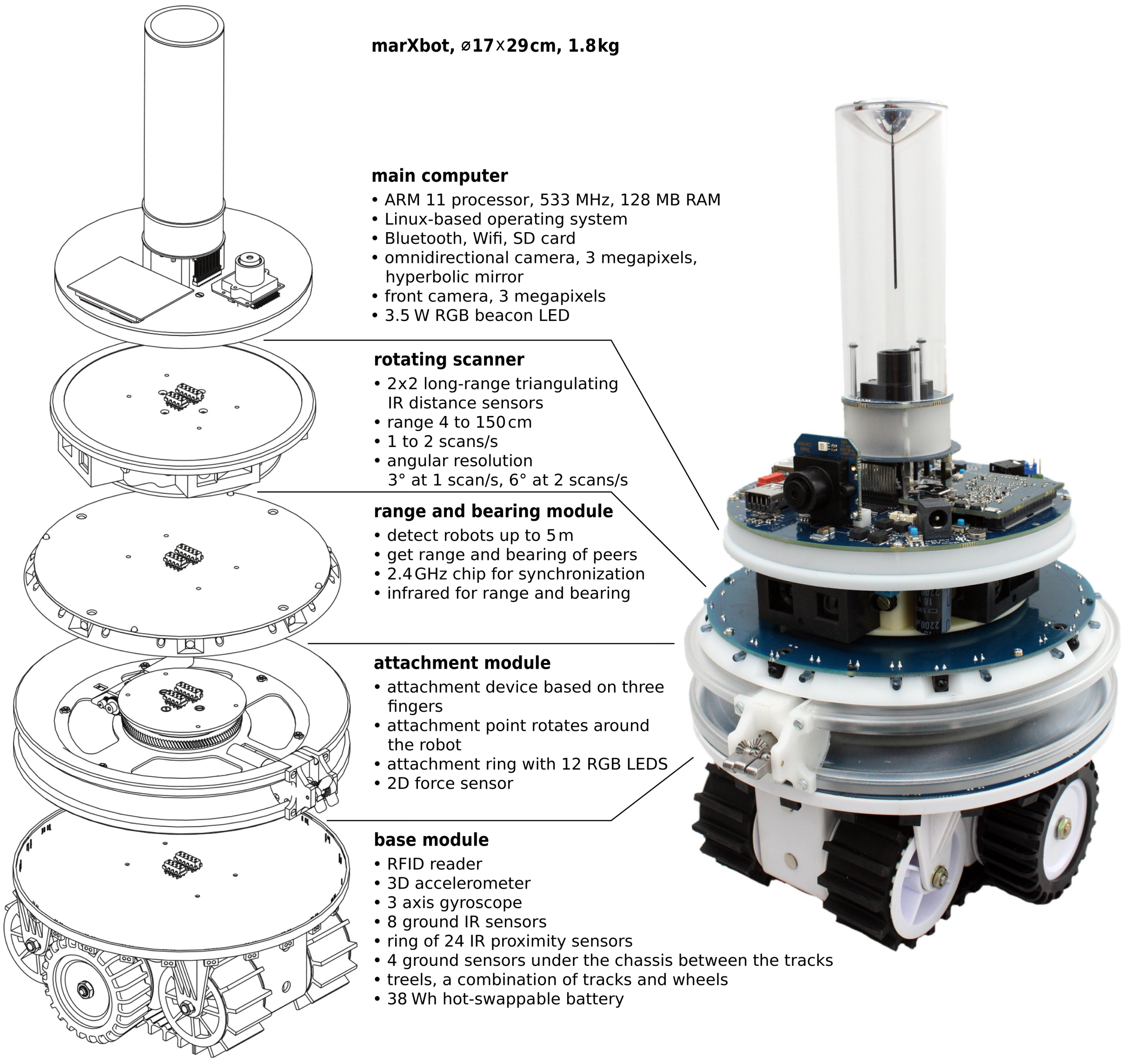
3. (Optional) Range and bearing module. A marXbot can get the relative position (distance + angle) of each individual inside a swarm using this module, up to a distance of 5 meters.

4. (Optional) Rotating scanner. This scanner is a low-cost version of common laser range scanners, using 4 infrared rangers (2 short range + 2 long range). It generates 480 points per second, at a rotation frequency of 1 to 2 Hz.

5. Main computer with front camera. The main computer is a Freescale i.MX31 processor (ARM 11, 533 MHz, 128 MB RAM), running a lightweight Linux distribution (no graphic desktop). A 3 megapixels camera can be mounted onto the board, in two different orientations (either upward or frontward).

6. (Optional) Upward camera and beacon LED. This module provides another 3 megapixels camera and a powerful RGB LED with built-in thermal protection. This camera is mainly used as an omnidirectional one.

[1] http://mobots.epfl.ch
[2] http://lsro.epfl.ch



marXbot, ⌀17x29cm, 1.8kg
main computer
• ARM 11 processor, 533 MHz, 128 MB RAM
• Linux-based operating system
• Bluetooth, Wifi, SD card
• omnidirectional camera, 3 megapixels, hyperbolic mirror
• front camera, 3 megapixels
• 3.5 W RGB beacon LED
rotating scanner
• 2x2 long-range triangulating IR distance sensors
• range 4 to 150 cm
• 1 to 2 scans/s
• angular resolution 3° at 1 scan/s, 6° at 2 scans/s
range and bearing module
• detect robots up to 5 m
• get range and bearing of peers
• 2.4 GHz chip for synchronization
• infrared for range and bearing
attachment module
• attachment device based on three fingers
• attachment point rotates around the robot
• attachment ring with 12 RGB LEDS
• 2D force sensor
base module
• RFID reader
• 3D accelerometer
• 3 axis gyroscope
• 8 ground IR sensors
• ring of 24 IR proximity sensors
• 4 ground sensors under the chassis between the tracks
• treels, a combination of tracks and wheels
• 38 Wh hot-swappable battery

Figure 1.1: Overview of the marXbot robot (source: [1])

1.2 Electronics of the Modules

The electronics of the modules is briefly described in Fig. 1.2. Each module has up to 3 dsPIC33 microcontrollers, a complete marXbot having a total of 10 microcontrollers. They are interconnected through a CAN bus. An I2C bus also exists and is used by the i.MX 31 to perform the power management tasks.

For this reason, the robot can't be operated without the computer module. This could be feasible, but solder jumps have to be soldered on the base module in this case. Ask the Mobots group for this modification.

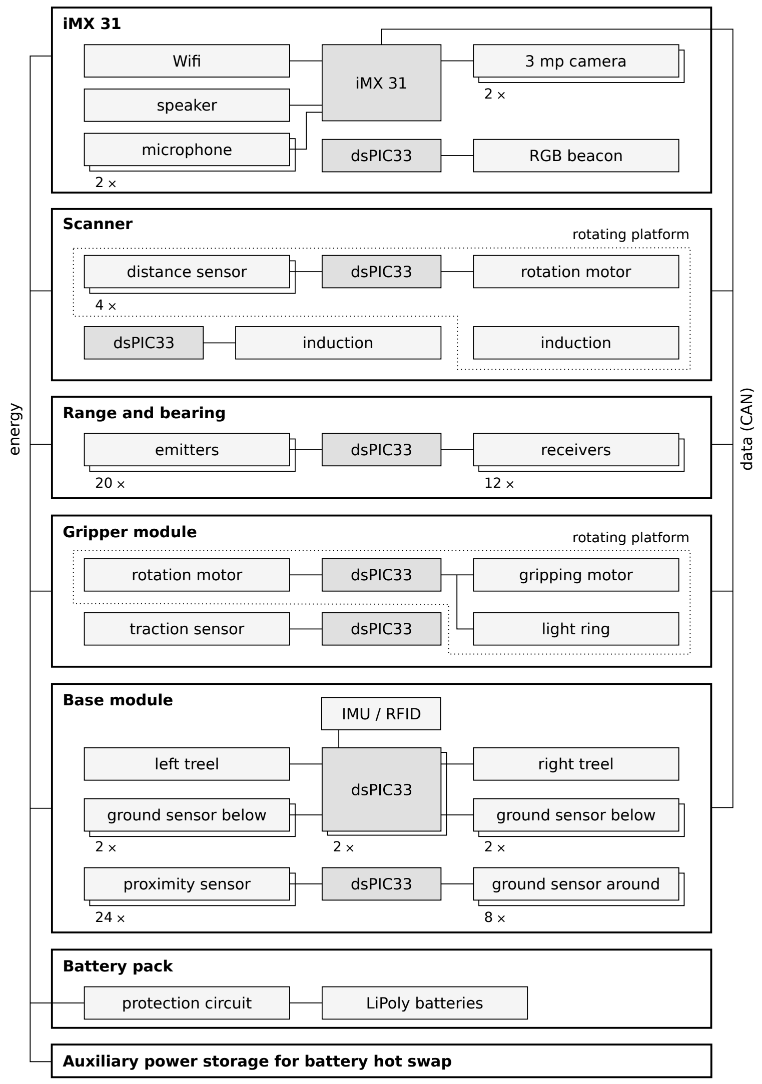


Figure 1.2: Overview of the electronics (source: [1]).

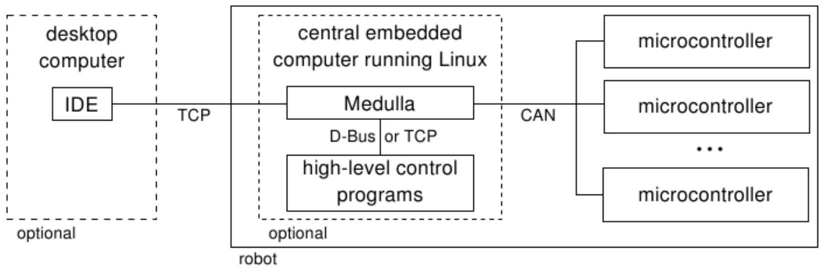
The CAN architecture is shown is Fig. 1.3 (a). This is a decentralized network, without any router. The embedded computer is also connected to this CAN network, through a CAN translator (not pictured) as the iMX31 has no built-in CAN interface. This embedded computer is connected to the outside world using a TCP/IP socket, operating either over Wi-Fi or USB (usbnet).

1.3 Software Description

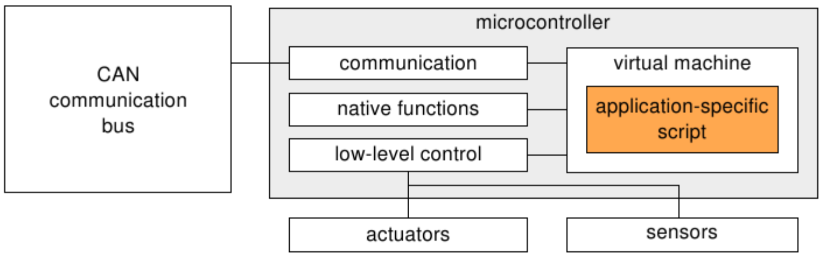
The great challenge in the marXbot is to manage the programs distributed between the 10 microcontrollers and the main CPU. Flashing all those units by hand would be untenable. This is why we take here advantage of the CAN bus, in order to flash remotely the microcontrollers.

Moreover, managing such a network only with low-level programming in C would be hard for the end-user. This assessment gave birth to ASEBA, a high-level, event-based scripting language. ASEBA scripting is only available on the microcontrollers. Thus, the program inside each microcontroller is structured in several layers (Fig. 1.3 (b)):

- A CAN bootloader is responsible for flashing the whole microcontroller, if needed[3].
- The low-level programming is done in C – with some optimizations performed in assembler – allowing good performances.
- The high-level behavior is done in ASEBA, leveraging the functions offered by the low-level layer[4]. The ASEBA script runs inside a small virtual machine. Each script is transferred through the CAN network and can be permanently written in Flash for persistency purpose.



(a) a typical ASEBA network in a robot



(b) a microcontroller in an ASEBA network

Figure 1.3: Architecture of the network (a) and microcontrollers (b) (source: [3]).

[3]In fact, two microcontrollers don't have a bootloader, and thus must be flashed using a Microchip ICD2 or ICD3: the internal translator (UART $\leftrightarrow$ CAN, not shown) on the computer module and the rotating dsPIC of the scanner module.
[4]Same apply here.

1.4 Basics

Here you will learn how to start with the marXbot. Please read carefully what follows.

1.4.1 Overview

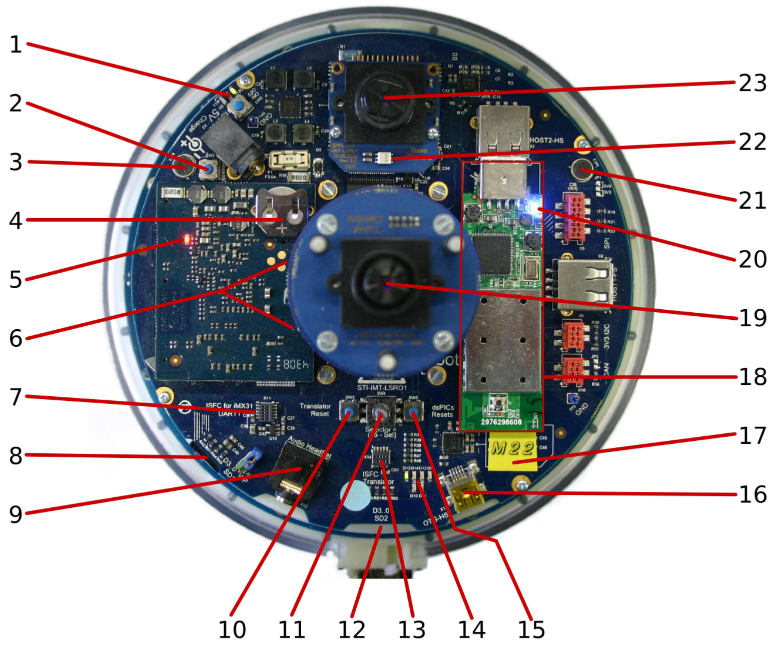


Figure 1.4: Top view of the marXbot.

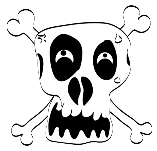
1. Power button + LED
2. iMX reset button
3. Mic 1
4. RTC backup battery
5. CoreOn LED
6. LED 0–3 (red) + LED 5 (RGB)
7. ISFC for iMX (UART1) – X11
8. MicroSD card
9. Audio jack
10. Translator reset button
11. Selector
12. SD card slot
13. ISFC for translator – X14
14. Translator status LEDs
15. dsPICs reset button
16. USB OTG (device)
17. MarXbot ID tag (Mxx)
18. Wi-Fi dongle
19. Upward camera
20. Wi-Fi status LED
21. Mic 2
22. LED 4 (RGB)
23. Front camera

1.4.2 Power on

Insert the battery inside the base module, with the contacts towards the back of the marXbot. The power LED (1) and core LED (5) will turn on. If nothing happens, press for 1 second on the power button (1). Shortly after, the heart-beating LED (6) will start to blink. After about 30 seconds, the LEDs of the translator (14) will also turn on, the marXbot is now ready to be used.

1.4.3 Power off

In order to power off the robot, press for 1 second on the power button (1) or execute the `poweroff` command on the robot. Wait until the robot shutdowns itself (all LEDs turned off), and then remove the battery. Please don't let the battery inside the robot, even if it is powered off.



Always properly shutdown the robot. The file system is of type EXT2 and is thus not journaled. A sudden power off could corrupt the file system, leaving you with no other chance than performing a `e2fsck` on the corrupted SD card. Most of the time, it will be easier to write a fresh root file system. See Sec. 3.5.2.

Always remove the battery. If you leave it inside, some capacitors may get charged through pull-up resistors, possibly powering on the microcontrollers and making the robot to jump down your table, like a Lemming would do. You have been warned.

1.5 Establishing a TCP/IP Connection

The marXbot is a versatile platform, and there are several ways to establish a connection.

1.5.1 Wi-Fi Connection

By default, the marXbot is configured to connect to the WPA-secured "mobots" network, using the embedded Wi-Fi dongle. This imply the availability of the corresponding access-point. The configuration is summarized in Sec. 1.7.1. The configuration of the robot can of course be changed in `/etc/network/interfaces` and `/etc/wpa_supplicant/`.

Supposing that you have the access-point, the LED on the Wi-Fi dongle will become blue when connected. As the USB driver on the marXbot has some problems, it may happen that the connection is not established properly, even 1 or 2 minutes after booting. In that case, remove carefully the dongle (18), insert it again and wait. If you are unlucky, you will have to perform this 2 or 3 times. As a last resort, reboot the robot using the power button (1).

On your computer, you also need to get associated with this private network, using the information given in Sec. 1.7.1. The IP address will be allocated by DHCP.

Using this Wi-Fi connection, the marXbot will have the static IP address

```
IP = 10.0.0.1xx
```

replacing `xx` by the number of your marXbot, written on the yellow sticker (17). If your computer is running zeroconf (avahi for example), you can use the hostname instead of the IP

```
marxbotxx.local
```

again replacing `xx` by the number of your marXbot.

1.5.2 USB Connection

In case the Wi-Fi connection is not available to you, you can create a network over USB using usbnet[5]. To do so, you will need a USB cable (mini-B to std-A). Plug the mini-B into the OTG-HS port (16) of the marXbot, while connecting the A plug to your computer. If you look at your network interfaces using `ifconfig`, you should see a new interface named `usb0`.

You first need to configure this interface. Your computer should be configured with a address in the subnet `192.168.0.1/24`, for example

```
sudo ifconfig usb0 192.168.0.1
```

Now, you should be able to ping the marXbot. They all have the same static IP address

[5]On Windows, you will have to perform a few more steps (not tested): http://docwiki.gumstix.org/index.php/Windows_XP_usbnet using the .inf provided here http://www.davehylands.com/linux/gumstix/usbnet/linux.inf

```
IP = 192.168.0.202
```

1.5.3 SSH Session

An SSH (Secure Shell) session will provide you with a remote console, established through a secured tunnel. Even if this security is not mandatory for common applications, it is a well standardized way. The following commands are given for a Linux or Mac OS host system. On Windows, you can use PuTTY as a graphical replacement.

Once the TCP/IP link is established, you can run a remote shell with the command

```
ssh root@marxbot_ip
```

It will ask you for the password, which is empty (just press the return key). If you don't want to be asked for the password each time, you can use public key authentication. The public key is already registered on the robot, you need to register the private key on your computer (to be done at each boot). The private key *mobots-ssh-key* is available on our website[6] . Run the command

```
ssh-add ~/.ssh/mobots-ssh-key
```

Copying files to the marXbot is easily achieved with the `scp` command

```
scp /path/to/file root@marxbot_ip:[/optional/path]
```

Don't forget the ":" when specifying the remote target, even if you omit the remote path (otherwise it will get copied locally). If you omit the path, it will copy the file in the working directory of the user (`/root/` in this case).

1.6 Establishing a Console Session

If no network is available, for example when experimenting with the Linux kernel or updating the marXbot, you will have to fall back on a console session. You will need an (emulated) serial port. This is the purpose of the "ISFC for iMX31 (X11)" connector (7) on the marXbot. Two modules have been developed. The following instructions are only given for a Linux-like operating system.

1.6.1 FTDI Module

This module (Fig. 1.5) allows you to establish a console session using a USB cable. The white side on the small male connector gives you the orientation of the pin number 1. It should match the corresponding mark on the marXbot's side.

Be sure not to invert the orientation.

When plugged in, you will see a new block device called `/dev/ttyUSBx`. You can also refer to the output of the `dmesg` command for the exact name.

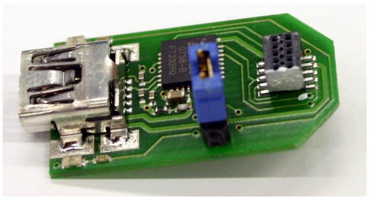


Figure 1.5: FTDI dongle.

We will use `minicom` as a serial terminal. To enter the configuration screen (only needed the first time), issue

[6] http://mobots.epfl.ch/mx31moboard/mobots-ssh-key

```
sudo minicom −s
```

and select “Serial port setup”. Enter the setup given in Table 1.1.

Table 1.1: Minicom configuration for USB

A - Serial Device	/dev/ttyUSB0
E - Bps/Par/Bits	921600 8N1
F - Hardware Flow Control	Yes
G - Software Flow Control	No

Before exiting, save the configuration as “ftdi”[7]. Launch again minicom, this time using

```
minicom ftdi
```

and it should connect.

1.6.2 Bluetooth Module

This module (Fig. 1.6) offers you a wireless serial link. The white side on the small male connector gives you the orientation of the pin number 1. It should match the corresponding mark on the marXbot’s side.

Be sure not to invert the orientation.

You first have to pair with the Bluetooth device before being able to use it.

```
hcitool scan
```

resulting for example in

```
Scanning ...
        00:26:68:0C:C2:6E        Nokia 5000d−2
        10:00:E8:6C:F0:52        LSRO1_LMX9838_D001
        00:21:AA:78:4C:FB        Nokia 6500s−1
        08:00:17:2D:09:B2        e−puck_0012
```

Copy the 12 digit in front of the device, “LSRO1_LMX9838_D001” in this example. Then

```
sudo rfcomm bind 0 10:00:E8:6C:F0:52
```

Don’t forget to replace `10:00:E8:6C:F0:52` with your current device. A new block device called `/dev/rfcomm0` should appear.

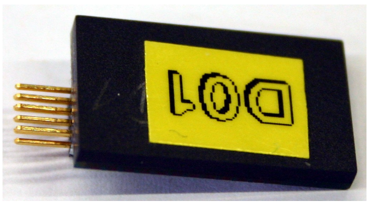


Figure 1.6: Bluetooth dongle.

Enter the configuration screen of `minicom`, exactly as for the FTDI dongle of the previous section, but using the parameters of Table 1.2.

Save the configuration under the name “rfcomm0” for example. You can then connect using

```
minicom rfcomm0
```

[7]The file is usually saved in `/etc/minirc.ftdi`

Table 1.2: Minicom configuration for Bluetooth

A - Serial Device	/dev/rfcomm0
E - Bps/Par/Bits	115200 8N1
F - Hardware Flow Control	No
G - Software Flow Control	No

1.7 Bonus

1.7.1 Configuration for the Mobots Network

The mobots wireless network is using only the 5 GHz band, to avoid the jam on the 2.4 GHz band. So be sure to use an appropriate access point and Wi-Fi dongle. In our group, we use the Linksys E3000 dual-band router, with the 2.4 GHz band disabled.

Table 1.3: WPA configuration.

SSID	mobots
Key type	WPA-PSK
Key	**Ask us**

Please, don't publish the WPA key, as it would compromise the security of the entire EPFL network. Thank you for your kind comprehension.

Table 1.4: IP configuration.

Router IP	10.0.0.1
Mask	255.255.255.0 (/24)
MarXbot IP range	10.0.0.100-199
DHCP range	10.0.0.200-249

Chapter 2

Introduction to Aseba Studio

As you saw in the introduction, the microcontrollers on the marXbot can load and execute ASEBA scripts. It is really easy to get started, you only need to install the tools on your computer and start playing. The main page of the ASEBA project is hosted on our website at the address `http://mobots.epfl.ch/aseba.html`. ASEBA is fully cross-platform and has been tested on Windows, Linux and MacOS.

This chapter is not about the ASEBA language. Please refer to the introductory material available on the official website, as well as to the API documentation in Chap. 6. An online documentation is also available in the Help menu of ASEBA Studio.

2.1 Getting Aseba

You first have to install ASEBA, either by installing the binaries or compile the sources. We recommend you to compile the sources downloaded from SVN, so you will be able to keep the most up-to-date version.

2.1.1 Getting the Binaries

You can download Windows binaries on the ASEBA website (`http://mobots.epfl.ch/aseba.html`). Binaries for Linux (.deb packages) are also available, but are unfortunately outdated. We will try to add more binaries and keep them up-to-date, but without any warrantee.

2.1.2 Compiling From Sources

The best solution is to build from sources. This is easily achieved thanks to CMake. You will need to install at least

- cmake
- ccmake

The SVN repository of ASEBA is hosted on gna[1]. ASEBA also depends on a number of other libraries:

- Dashel[2]
- Enki[3]
- Qt4
- Boost
- Qwt
- SDL

Dependencies other than enki and dashel should be available through the package manager of your favorite distribution. Take care to install the `-devel` packages also. For example, on Fedora 14

[1]`http://gna.org/projects/aseba`
[2]SVN repository: `http://gna.org/projects/dashel`
[3]SVN repository: `http://gna.org/projects/enki`

```
sudo yum install qt−devel boost−devel qwt−devel SDL−devel
```

Dashel and enki are to be download on gna using SVN and compiled using CMake (should be easy if other dependencies are correctly installed)

```
cmake .
make
sudo make install  (optional)
```

The last step is to compile ASEBA. The only tricky step is to correctly setup the path to the dependencies. Most of them should be automatically detected, but you will have to specify by hand the ones for dashel and enki (if not installed). To configure the paths, run

```
ccmake .
```

If the cache is empty, press 'c' to configure, then 'e' when errors are displayed. You should then be able to enter the paths by pressing 'enter'. Typical paths are given in Table 2.1, adapt them to your configuration. When done, you can rerun the configuration by pressing 'c' or executing `cmake`. When the configuration succeeds, compile as before

```
cmake .
make
```

Table 2.1: An example of CMake configuration for ASEBA

CMAKE_INSTALL_PREFIX	/usr/local
DASHEL_INCLUDE_DIR	/home/vaussard/svn/dashel
DASHEL_LIBRARY	/home/vaussard/svn/dashel/libdashel.a
ENKI_INCLUDE_DIR	/home/vaussard/svn/enki
ENKI_LIBRARY	/home/vaussard/svn/enki/enki/libenki.a
ENKI_VIEWER_LIBRARY	/home/vaussard/svn/enki/viewer/libenkiviewer.a
QT_QMAKE_EXECUTABLE	/usr/bin/qmake-qt4
QWT_INCLUDE_DIR	/usr/include/qwt
QWT_LIBRARIES	/usr/lib64/libqwt.so
SDLMAIN_LIBRARY	/usr/lib64
SDL_INCLUDE_DIR	/usr/include/SDL
SDL_LIBRARY	/usr/lib64/libSDL.so;-lpthread

2.2 Running Aseba Studio

The first step is to establish a TCP/IP network with the robot. Follow the instructions given in Sec. 1.5, either using Wi-Fi or USB (usbnet).

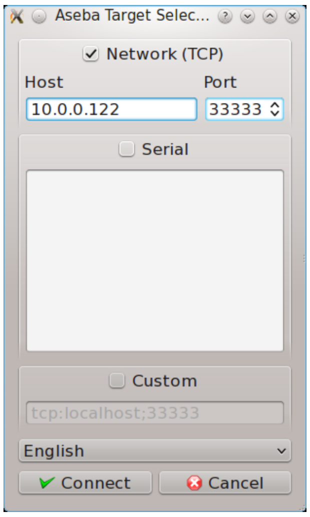
Go into the folder where you have compiled ASEBA, and run the `asebastudio` executable

```
./studio/asebastudio
```

The connection screen of Fig. 2.1 will show up. Enter the corresponding IP address or hostname, as described in the introduction. *Port* should be 33333. If successful (try to ping the robot otherwise), the ASEBA Studio GUI will appear (Fig. 2.2).

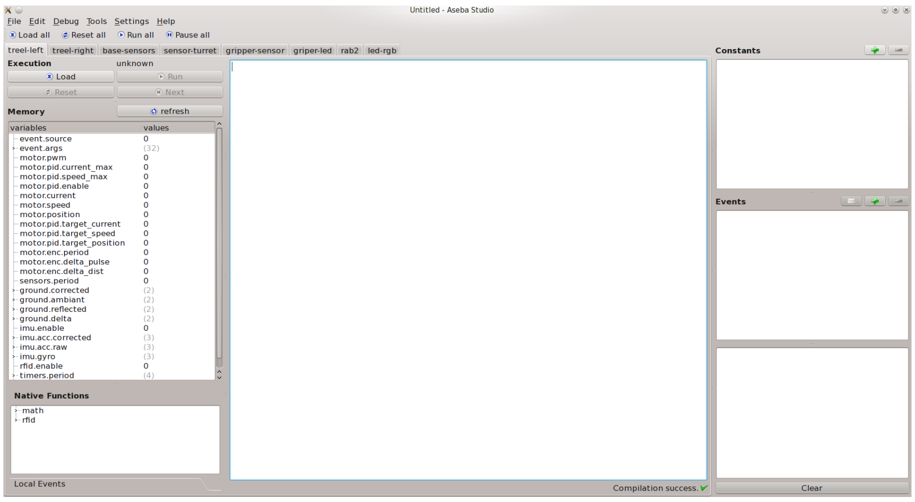
Each ASEBA node (→ each microcontroller) appears in a separate tab. The full documentation for each node is available in Chap. 6. For a full marXbot, the following nodes are available:

- *treel-left*: Left wheel and corresponding sensors.
- *treel-right*: Right wheel and corresponding sensors.
- *base-sensors*: Range sensors in the base module.
- *sensor-turret*: Rotating scanner module.
- *gripper-sensor*: Sensors for the gripper module.
- *gripper-led*: Gripper's RGB LEDs and actuation.



Host: 10.0.0.122
Port: 33333
tcp:localhost;33333
English

Figure 2.1: ASEBA Studio connection screen.


Untitled - Aseba Studio
File Edit Debug Tools Settings Help
Load all Reset all Run all Pause all
treel-left treel-right base-sensors sensor-turret gripper-sensor griper-led rab2 led-rgb
Execution
Load
Run
Reset
Next
Memory
refresh
Native Functions
math
rfid
Local Events
Constants
Events
Clear

Execution: unknown

variables	values
event.source	0
event.args	(32)
motor.pwm	0
motor.pid.current_max	0
motor.pid.speed_max	0
motor.pid.enable	0
motor.current	0
motor.speed	0
motor.position	0
motor.pid.target_current	0
motor.pid.target_speed	0
motor.pid.target_position	0
motor.enc.period	0
motor.enc.delta_pulse	0
motor.enc.delta_dist	0
sensors.period	0
ground.corrected	(2)
ground.ambiant	(2)
ground.reflected	(2)
ground.delta	(2)
imu.enable	0
imu.acc.corrected	(3)
imu.acc.raw	(3)
imu.gyro	(3)
rfid.enable	0
timers.period	(4)

Compilation success.

Figure 2.2: ASEBA Studio.

- *rab2*: Range and bearing module.
- *led-rgb*: Power RGB LED.

2.3 Your First Scripts

This section will provide you some basic examples for your first ASEBA scripts.

2.3.1 Light On the RGB LED

Select the *led-rgb* tab and enter the following piece of code

```
call led.rgb(500,0,0)
```

The compilation into bytecodes is done on-the-fly. Now, you just need to press the *Load* and *Run* buttons on your left. The power RGB LED will turn red! If you prefer green, write instead

```
call led.rgb(0,500,0)
```

Load, run and the LED is now green! This powerful LED turns out to heat quite a lot. This is why a built-in thermal protection has been added. You can measure the LED's temperature by enabling the sensor

```
led.temperature.period=100
call led.rgb(0,500,0)
```

Load, run and press the *Refresh* button. You will see the measured value in the `led.temperature` variable. According to the documentation of Sec. 6.2.8, the unit is 0.1 °C. If you increase the intensity or enable several colors at the same time, you will see this value increasing each time you press the *Refresh* button.

2.3.2 Your First Local Event

Following up the first example, we will slightly increase the complexity in order to make the LED blink at a frequency of 0.5 Hz, using a local event. Still using the *led-rgb* tab, copy the Listing 2.1.

```
# Declare variables
var time = 0
var state = 0
var value

# Set the sampling and timer0 period
led.temperature.period = 100
timers.period[0] = 100

# timer0 event routine
onevent timer0

time = time + 1
if time == 10 then
    time = 0
    state = ~state
    value = 500*state
    call led.rgb(value,0,0)
end
```

Listing 2.1: Second example: blinking LED

A few words of explanation:

- Oneline comments begin with #.
- All the variables are global and should be declared at the beginning.
- Local events are fired by the microcontrollers and cached locally. Each microcontroller has its own set of local events. They are displayed using the *Local Event* tab at the lower left corner of the GUI.
- `timers.period[0]` sets the period (in milliseconds) for the local event fired by Timer0.
- `onevent timer0` is the routine executed when the Timer0 event is fired.

2.3.3 Your First Global Event

Global events can be exchanged between the microcontrollers connected to the CAN bus, and also to / from a computer connected on the same network, like with asebastudio. We will demonstrate this using the proximity sensors, coupled with the RGB LED to display the detection of an obstacle. The first script, shown in Listing 2.2, is to be coded in the *base-sensors* node. In addition, you will have to add a constant named THRESHOLD set to 1000 (upper right corner). The global events are also to be added in the *Events* box, on the right (see Fig. 2.3). The number next to the event is the number of arguments this event will take. In our case, it is zero.

The code is rather self-explaining, with the help of the API documentation of Chap. 6. The sensors are refreshed every 100 ms, and the update function tests each sensor for a value greater than THRESHOLD. If such a case exists, the global event *obstacle* is emitted.

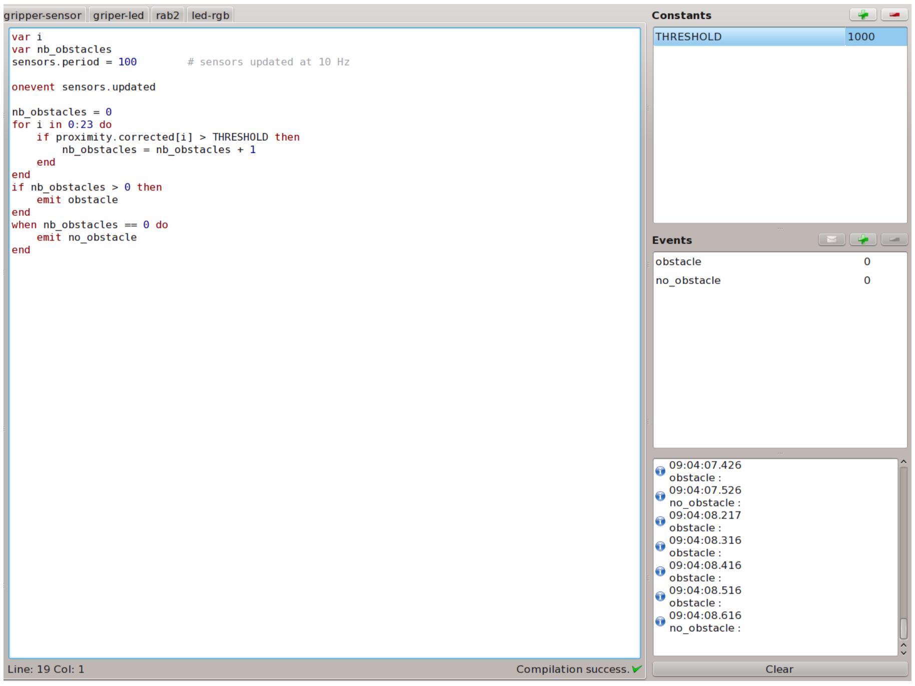
You can already play with this part. Load and execute the script. You will see fired events in the log window, on the bottom right corner, each time you approach your hand from the base sensors (*obstacle* events) and when you go back (*no_obstacle* events).

```
var i
var nb_obstacles
sensors.period = 100          # sensors updated at 10 Hz

onevent sensors.updated

nb_obstacles = 0
for i in 0:23 do
    if proximity.corrected[i] > THRESHOLD then
        nb_obstacles = nb_obstacles + 1
    end
end
if nb_obstacles > 0 then
    emit obstacle
end
when nb_obstacles == 0 do
    emit no_obstacle
end
```

Listing 2.2: Third example: proximity sensors - *base-sensors* node



```
var i
var nb_obstacles
sensors.period = 100        # sensors updated at 10 Hz

onevent sensors.updated

nb_obstacles = 0
for i in 0:23 do
    if proximity.corrected[i] > THRESHOLD then
        nb_obstacles = nb_obstacles + 1
    end
end
if nb_obstacles > 0 then
    emit obstacle
end
when nb_obstacles == 0 do
    emit no_obstacle
end
```

Line: 19 Col: 1 Compilation success.

Constants

THRESHOLD	1000

Events

obstacle	0
no_obstacle	0

```
09:04:07.426
obstacle :
09:04:07.526
no_obstacle :
09:04:08.217
obstacle :
09:04:08.316
obstacle :
09:04:08.416
obstacle :
09:04:08.516
obstacle :
09:04:08.616
no_obstacle :
```

Figure 2.3: ASEBA Studio during third example.

You can then catch those events from the *led-rgb* node, for example to light on the LED. Copy the script of Listing 2.3 in the corresponding tab.

```
onevent obstacle
call led.rgb(300,0,0)

onevent no_obstacle
call led.rgb(0,0,0)
```

Listing 2.3: Third example: proximity sensors - *rgb-led* node

2.4 Advanced Aseba Topics

FIXME

2.5 Other Aseba Tools

ASEBA is not limited to ASEBA Studio. There are several other (command-line) tools, which can be useful in several situations. We will give a brief summary in this section. Please refer to the output of the `--help` switch for detailed help on each command.

2.5.1 Dashel Targets

The following commands use so-called "dashel targets", which are strings describing the interface to connect to. They are to be written between quotation marks " ". A summary is given in Table 2.2. If you want to connect to several dashel targets at the same time, you can separate them with with a white space (␣), like

```
asebaswitch "ser:device=/dev/ttyUSB0 tcp:10.0.0.154"
```

Table 2.2: Dashel targets.

Target type	Dashel string
Serial	"ser:device=/dev/ttyXX;fc=hard;baud=921600"
TCP / IP	"tcp:10.0.0.1XX;port=33333"

The IP can be replaced by `localhost`, for example when connecting to `asebaswitch` or `asebamedulla`.

2.5.2 asebacmd

`asebacmd` is useful for sending commands over the ASEBA network. Among the commands, the most useful are summarized in Table 2.3.

Table 2.3: Useful commands for `asebacmd`.

Command	Usage
`presence`	Broadcast an identification request.
`usermsg [type] [length]`	Send a global event on the network of ID `type` with `length` number of data. For now, the (dumb) data are 0 1 2 ... `length`-1.
`sb [nodeid]`	Reset the node with ID `nodeid` (number).

2.5.3 asebaswitch – asebamedulla

`asebaswitch` and `asebamedulla` are almost identical. Both of them are used to connect several ASEBA components together, like a control software, a dump program and an ASEBA network. This will be explained in greater details in Chap. 5. `asebaswitch` supports all the dashel targets (serial, TCP). Moreover, `asebamedulla` can connect components using D-Bus.

The only limitation is the ID of the nodes[4]. Each ID must be unique inside the same network. This implies that two marXbots can't connect to the same ASEBA network, as this would give rise to ID conflicts.

If you want to leverage D-Bus using your own program and `asebamedulla`, you can look at the example Python script, located in the `medulla` folder. The system bus is used on the marXbot, while the session bus is used on a remote computer.

For both programs, you can use the `-d -v` switches in order to display what is going on the network, which can be quite useful for troubleshooting.

[4]The ID of a node can be found in ASEBA Studio by showing the hidden variables.

2.5.4 asebadump

`asebadump`, connected to an `asebaswitch`, is used to dump all the data going on the ASEBA network. Those data are output on the console, using a raw format.

2.5.5 asebarec – asebaplay

`asebarec`, connected to an `asebaswitch`, is used to record global events on the ASEBA network. Those events are time-stamped and are in a human-readable format.

`asebaplay` is used to inject recorded events into the ASEBA network. You can choose to play them faster than usual.

2.5.6 asebaeventlogger

`asebaeventlogger` is a Qt interface used to plot a specific event over time. However, this can be also achieved in ASEBA Studio, so it's of little use.

Chapter 3

Advanced Topics

This chapter will cover more advanced usages, like accessing the camera, the LEDs or updating the robot.

3.1 Using the Camera

Up to two Aptina MT9T031 cameras can be mounted on the robot. However, only one acquisition bus is available on the current processor, thus only one camera can be used at the same time. This can be partly workarounded, as we will see in Sec. 3.1.4.

The camera appears in Linux as a block device `/dev/video0`. Operations should be carried on this node.

3.1.1 Selecting the Camera

The selection between the two cameras is done by an external electronics, activated by one of the GPIO (General Purpose IO) pin of the i.MX processor. This pin is available in user-space and is called `gpio28`.

To select the upward camera[1] (19)

```
echo 0 > /sys/class/gpio/gpio28/value
```

If you prefer to use the frontward camera (23)

```
echo 1 > /sys/class/gpio/gpio28/value
```

Never change this configuration while the node is opened. Always close the port prior to any modifications. In fact, this could mislead the driver, as both the cameras share the same bus. You must also wait 100 μs between a change on the GPIO and the opening of the node.

3.1.2 Streaming Images

Streaming images to your computer is the most straightforward operation. First, be sure *gstreamer* is installed on your computer. This is easily achieved through the packager manager of your favorite Linux distribution.

Then, on the robot, launch `camera-viewer`, which is a custom software installed by default.

```
./camera-viewer -qws -display VNC:0
```

Then, on your computer

```
gst-launch-0.10 tcpclientsrc host=10.0.0.1xx port=11111 ! jpegdec ! autovideosink
```

Set the IP address to the one of your marXbot (see Sec. 1.5).

You can also set the camera's parameters (exposure and gain), as `camera-viewer` offers this facility by exporting a VNC remote desktop. To access it, use your favorite VNC client (for example KRDC on KDE desktops) using the IP of your marXbot.

[1] This is the default option.

3.1.3 Coding

If you wish to access the camera directly, the best way is to process the images on the iMX. Any program can get access to the camera, using the Video4Linux2 API. The documentation of this API is unfortunately quite messy. The `camera-viewer` application is a good start, ask us for the source.

3.1.4 Advanced

As said earlier, two connectors are available for connecting up to two cameras at the same time. However, there is only one acquisition bus, shared between the two cameras. Thus, we strongly recommended to use only one camera at the same time.

With some tricks, it is however possible to access both alternatively, but this is strongly discouraged and won't be supported.

Now stop joking, it is really `really` **really** discouraged...

To do so, you will need to write your own acquisition program (previous section), in order to synchronize the operations. Follow the following steps.

1. Select the first camera.
2. Wait at least 100 μs.
3. Open and configure the node.
4. Select the other camera.
5. Wait at least 100 μs.
6. Close.
7. Open and configure the node with the same configuration.
8. Now you should be able to alternate the acquisition on both cameras, by changing the GPIO and waiting between each acquisition. Don't change the GPIO in the middle of an acquisition, really...

3.2 GPIOs and LEDs

On the marXbot, GPIOs (General Purpose Input Output) and LEDs are exported by the kernel. This is a convenient way to interact with the hardware level.

3.2.1 GPIOs

GPIOs can be accessed on the marXbot through the `/sys/class/gpio/*` pseudo-nodes. Table 3.1 shows available GPIOs and their usage.

An output GPIO can be put high with

```
echo 1 > /sys/class/gpio/??/value
```

and put low with

```
echo 0 > /sys/class/gpio/??/value
```

For input GPIOs, you can read the value using

```
cat /sys/class/gpio/??/value
```

Table 3.1: Available GPIOs

Name	Direction	Function
gpio21	Output	Translator reset
gpio22	Output	dsPICs reset
gpio28	Output	Select the camera
gpio40	Input	Selector bit 0
gpio41	Input	Selector bit 1
gpio42	Input	Selector bit 2
gpio43	Input	Selector bit 3
gpio87	Input	Battery detection

3.2.2 LEDs

LEDs can be accessed on the marXbot through the `/sys/class/leds/*` pseudo-nodes. Table 3.2 shows available LEDs and their location. For each node, the actual brightness is retrieved with

```
cat /sys/class/leds/??/brightness
```

and the maximum brightness with

```
cat /sys/class/leds/??/max_brightness
```

while the brightness is set with

```
echo brightness > /sys/class/leds/??/brightness
```

Replace `brightness` with the brightness you desire.

Triggers can also be set. This way allows you to automatically turn on / off the LEDs, based on some system events. A list of available triggers for each LED is obtained with

```
cat /sys/class/leds/??/trigger
```

and a trigger can be set (for example heartbeat)

```
echo heartbeat > /sys/class/leds/??/trigger
```

Table 3.2: Available LEDs

Name	Location
ar9170-phy2::assoc	Wi-Fi dongle (color?)
ar9170-phy2::tx	Wi-Fi dongle (color?)
coreboard-led-0:red:running	iMX processor board (heart-beat)
coreboard-led-1:red	iMX processor board
coreboard-led-2:red	iMX processor board
coreboard-led-3:red	iMX processor board
coreboard-led-4:red	Front camera (RGB → R)
coreboard-led-4:green	Front camera (RGB → G)
coreboard-led-4:blue	Front camera (RGB → B)
coreboard-led-5:red	iMX processor board (RGB → R)
coreboard-led-5:green	iMX processor board (RGB → G)
coreboard-led-5:blue	iMX processor board (RGB → B)

3.3 Power Management

You can read the battery voltage on the marXbot using the command:

```
qdbus --system ch.epfl.mobots.power /battery ch.epfl.mobots.power.battery.GetVoltage
```

3.4 Accessing the Root Filesystem and Flash

The files on the marXbot are stored either on the micro-SD card (root filesystem), or in the embedded NOR memory (Linux kernel). The way to access those files is highly depend on the storage media.

3.4.1 Micro-SD memory

The filesystem on the micro-SD card can be easily accessed either through the network (Sec. 1.5), or directly by removing the SD card (halt the robot before proceeding) and combined with a card-reader.

3.4.2 NOR memory

The embedded NOR memory has the following layout.

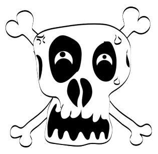
Table 3.3: Typical NOR layout (may change).

Node	Name	Size [kB]	Read-only
mtdblock0	RedBoot	256	No
mtdblock1	kernel	2048	No
mtdblock2	FIS directory	124	Yes
mtdblock3	RedBoot config	4	Yes

Only the Linux kernel and the bootloader partition is writable. The other partitions are forced to read-only.

Please, don't mess the bootloader partition.

3.5 Updating the Robot



There are tons of ways to screw up the update. So don't do it unless needed. If you have any doubt or any potential misunderstanding, please, ask before performing the update.

3.5.1 Simple Update

If the robot has access to the Internet using the Wi-Fi, this simple update process is enough. It will update everything, including

- The Linux distribution
- The Linux kernel on the NOR
- The firmware of the microcontrollers[2]

Using a remote console (SSH or serial), issue the commands

```
opkg update
opkg upgrade
```

If you want to update the microcontrollers' firmware, first make sure the battery is not empty. Then

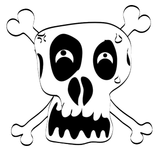
```
update−robot
```

[2]Excepted the 2 microcontrollers without a bootloader, see Sec. 1.3

DO NOT, for *ANY REASON* kill this task (or do ctrl-c).

3.5.2 The Hard Way

This way may be your only chance, if the distribution on the robot is really old (without the Wi-Fi support) or if your SD card is corrupted.



Do this only, only, only if needed. Ask us if you are unsure. This can leave the robot unusable if you screw up the kernel and we will have to load a new kernel by hand.

Updating the SD Card First download the script located here `http://mobots.epfl.ch/mx31moboard/create_sd.sh`. Change the permissions to allow its execution[3]. This script assume that your SD card is already properly formatted. Insert the SD card into your computer. The following commands, executed from the directory where you have downloaded `create_sd.sh`, will help you to download the latest available root filesystem. The tarball must be renamed to `img.tar.bz2` for the script to work.

```
wget http://mobots.epfl.ch/mx31moboard/latest_rootfs
wget -i latest_rootfs
FILE="ls_Angstrom*"
mv $FILE img.tar.bz2
 install -d mnt
sudo ./create_sd footbot MARXBOT_NUMBER /dev/partition_of_the_sd_card
```

replacing MARXBOT_NUMBER by the two-digits number of the marXbot (yellow sticker) and /dev/partition_of_the_sd_card by the device of the SD card (usually something like [FIXME]). This will format the SD card, copy the image and then do some configuration stuff.

A few important points:

- Keep the marXbot number correct. This is important to keep track of each robot and to assign the correct IP address.
- You must use the first partition of the SD card like [FIXME] (the SD cards on the robots are already correctly partitioned for this), not the whole device ([FIXME]).

Updating the Kernel To update the kernel, boot it with

- The new SD card
- A charged battery
- The Bluetooth module on the X11 connector (see Sec. 1.6)

Plug the battery in the robot, connect to the Bluetooth and wait until the robot displays the login prompt (it can take some time for the first boot).

Log in as root (empty password) and do

```
cat /boot/zImage > /dev/mtdblock1
```

When the command is done (it will take some time, writing inside the NOR memory is slow), reboot

```
reboot
```

It should now be able to connect to the Wi-Fi or usbnet. If it does not connect to the Wi-Fi, reboot until it succeeds (wait at least 1 minute after each boot to wait for the connection).

Now, you can proceed with the regular update process as explained in Sec. 3.5.1.

[3] `chmod +x create_sd.sh`

Chapter 4

Introduction to iMX Programming

Programs can be developed for the embedded computer. This allows you for example to [FIXME]. However, this embedded processor is based on an ARM architecture, implying the use of a cross-compiler to generate the binaries. This part can only be performed on a Linux host system, as the SDK is not available for other operating systems.

4.1 The SDK

The SDK includes several pieces of software, so you can develop, investigate and debug programs running on the ARM processor. This includes

- The cross-gcc toolsuite (`arm-angstrom-linux-gnueabi-gcc` and so on...)
- The CMake and Qt tools (`cmake`, `qmake`,...)
- Cross-compiled standard libraries (`libc`, `libstdc++`,...)
- Other cross-compiled libraries you will need to link most of the programs we provide ()

4.1.1 Installing the SDK

To find out the latest version of the toolchain, first download the manifest from our website

```
wget http://mobots.epfl.ch/mx31moboard/sdk/latest_sdk
```

and use the link provided in the file to download the SDK corresponding to the architecture of your development computer. For example, with a 64 bits computer

```
wget http://mobots.epfl.ch/mx31moboard/sdk/angstrom-2010.4-test-20100608-x86_64-linux-
    armv6-linux-gnueabi-toolchain-mobots.tar.bz2
```

If you want to download the SDKs for all architectures

```
wget -i latest_sdk
```

Once done, you can install it on your development machine. To install it, just decompress the archive at the root of your computer

```
sudo tar -xjf toolchain.tar.bz2 -C /
```

It will install its files in `/usr/local/angstrom/arm`. You then need to source the `environment-setup` script at startup, for example from your `.bashrc`, in order to set a number of environment variables for the SDK to work. To achieve this, just execute once

```
echo source /usr/local/angstrom/arm/environment-setup >> ~/.bashrc
```

4.1.2 Using the Toolchain

You can then invoke the cross-GCC using the *arm-angstrom-linux-gnueabi-* prefix. In order to make your life easier, a CMake file is freely available at `http://mobots.epfl.ch/mx31moboard/sdk/moboard.cmake`. It will set the necessary variables for a smooth compilation, like this:

```
cmake -DCMAKE_TOOLCHAIN_FILE=~/moboard.cmake .
```

Once compiled, simply copy the binary to the target and run it.

4.2 Remote Debugging

The toolchain includes `gdb` (called `arm-angstrom-linux-gnueabi-gdb`) and it can be used to debug programs running on the marXbot from your computer. To achieve this, we use remote cross-target debugging with `gdbserver`.

First you need to compile your program with the debug option (-g with gcc) so that it contains the debug symbols. With CMake, you need to add the -DCMAKE_BUILD_TYPE=Debug option. If we go back to the above command, it becomes:

```
cmake -DCMAKE_BUILD_TYPE=Debug -DCMAKE_TOOLCHAIN_FILE=~/moboard.cmake .
```

On the marXbot, run your program with `gdbserver` and the port the server is going to listen to for the TCP connection:

```
gdbserver :5000 yourprogram
```

On your computer, just start your program with the toolchain's debugger:

```
arm-angstrom-linux-gnueabi-gdb
```

If `gdb` complains about not being able to load libexpat.so.0, just create a link /usr/lib/libexpat.so.0 to /usr/lib/libexpat.so.

First, you need to tell `gdb` where the libraries are on your computer. Here is the default SDK install path:

```
(gdb) set solib-search-path /usr/local/angstrom/arm/arm-angstrom-linux-gnueabi
```

Then tell `gdb` to read the symbols from your debug enabled program:

```
(gdb) symbol-file yourpgram
```

Then you just need to issue the target command in `gdb` on your computer so that it connects to the `gdbserver` on the robot:

```
(gdb) target remote marXbot_ip:5000
```

From this point, you can use `gdb` as if you were debugging on your computer. Please note that not all libraries have the debug symbols enabled.

Chapter 5

Control Architecture

Chapter 6

Aseba API

6.1 Calibration

The marXbot should already be calibrated.

Any new marXbot must be calibrated prior to first use. To do so, use the relevant python script. This script uses `asebamedulla` and thus depends on `D-Bus` to run. A recent Linux distribution should run `D-Bus` by default. The procedure to follow will be displayed by the python script. Make sure to do the calibration in an environment without external infrared sources (such as windows exposed to sun, lamps, etc).

6.2 Aseba Application Programmer Interface

6.2.1 Introduction

All the marXbot modules have common interfaces available on any module and some specific to each modules. The specific interfaces will be discussed in each module's subsection.

Visible and Hidden API

The API can be subdivided in two major parts:

Visible Part
This subset of the API is safe to use. Some range checking is done on the variables and natives functions. It *should* not be possible to crash the microcontroller when using it.

Hidden Part

This subset is unsafe to use blindly. You *must* double-check what you are doing when using such variables or natives functions. You *can* crash the microcontroller, burn it or damage the whole robot.

Such variables or natives functions are hidden by default in ASEBA Studio[1]. These begin with a "_" or have "_." in their names and will be displayed in gray in the rest of this manual.

Conventions

Some natives functions and variables uses angles. In ASEBA, the angles are 16 bit integers where -32768 is equal to $-\pi$, 0 is a null angle, and 32767 is almost equal to π.

In the documentation of natives functions, variables in capitals letters denote vectors, while variables in minuscules denote scalars. Variables in capitals letters with indices, such as A_i means for all i in the range of indices of A.

[1]You can change this by checking the `Settings`→`Show hidden variables and functions` option

Settings

A hidden functionality of ASEBA is the settings subsystem. In almost every microcontrollers some calibration constants must be stored across powerdown. These settings should not be modified without special care. They are stored inside the flash of the microcontroller and thus only a limited number of write cycle is allowed.

6.2.2 Common API

Variables

`event.source`
: The ASEBA node identifier of the incoming event. Only valid for external events. If you write it, you loose the identifier of the last received event.

`event.args`
: The arguments of the incoming event. Only valid for external events. If you write it, you loose the arguments of last received event.

Hidden variables

`_id` The ASEBA node identifier of the current virtual machine. Unsafe to write.

`_fwversion`
: The firmware's version of the microcontroller. Unsafe to write.

Events

There is no common events.

Natives Functions

`math.copy(A,B)`
: Copy the vector B in the A vector, element by element: $A_i = B_i$.

`math.fill(A,c)`
: Fill each element of the A vector by the constant c: $A_i = c$.

`math.addscalar(A, B, c)`
: Compute $A_i = B_i + c$ where c is a scalar.

`math.add(A, B, C)`
: Compute $A_i = B_i + C_i$ where A, B and C are three vectors of the same size.

`math.sub(A, B, C)`
: Compute $A_i = B_i - C_i$ where A, B and C are three vectors of the same size.

`math.mul(A, B, C)`
: Compute $A_i = B_i \cdot C_i$ where A, B and C are three vectors of the same size.

Warning: This is not a dot product.

`math.div(A, B, C)`
: Compute $A_i = B_i / C_i$ where A, B and C are three vectors of the same size.

Note: An exception will be trigged if a division by zero occurs.

`math.min(A, B, C)`
: Write the minimum of each element of B and C in A where A, B and C are three vectors of the same size: $A_i = \min(B_i, C_i)$.

`math.max(A, B, C)`
: Write the maximum of each element of B and C in A where A, B and C are three vectors of the same size: $A_i = \max(B_i, C_i)$.

`math.dot(r, A, B, n)`
: Compute the dot product between two vectors of the same size A and B:

$$r = \frac{\sum_i (A_i \cdot B_i)}{2^n}$$

`math.stat(V, min, max, mean)`
: Compute the maximum, the mean and the minimum value of vector V.

`math.muldiv(A, B, C, D)`
: Compute multiplication-division using internal 32 bit precision: $A_i = \frac{B_i \cdot C_i}{D_i}$.

Note: An exception will be trigged if a division by zero occurs.

`math.atan2(A, Y, X)`
: Compute $A_i = \arctan\left(\frac{Y_i}{X_i}\right)$ using the signs of Y_i and X_i to determine the quadrant of the output, where A, Y and X are three vectors of the same size.

Note: $X_i = 0$ and $Y_i = 0$ will produce $A_i = 0$.

`math.sin(A, B)`
: Compute $A_i = \sin(B_i)$ where A and B are two vectors of the same size.

`math.cos(A, B)`
: Compute $A_i = \cos(B_i)$ where A and B are two vectors of the same size.

`math.rot2(A, B, angle)`
: Rotate the vector B by *angle*, write result to A.

Note: A and B must be two vectors of size 2.

`math.sqrt(A, B)`
: Compute $A_i = \sqrt{B_i}$ where A and B are two vectors of the same size.

`math.nzseq(a, B, m)`
: Write to a the middle index of the largest sequence of non-zero elements from B, -1 if not found or if the sequence is smaller than m.

Hidden Natives Functions

`_system.reboot`
: Hard-reset the microcontroller immediately. The microcontroller will execute the bootloader immediately after the reset.

`_system.settings.read(a, v)`
: Read the setting number a and put its value in v.

`_system.settings.write(a, v)`
: Write the setting number a with value v.

`_system.settings.flash`
: Flash the settings inside the microcontroller's flash.

Warning: This is really dangerous as the flash wears every time you write to it.

Settings

There is no common settings.

6.2.3 Standard Motor Module

This is the interface used by most motors on the marXbot. It comprises the set of variables, events and natives functions which manage the PID controllers of the motors. Each motor has his own name and there can be multiple motors on the same ASEBA node. Thus the variables, natives functions and event names are prefixed by the motor's name. We will use the following name in this section: "M".

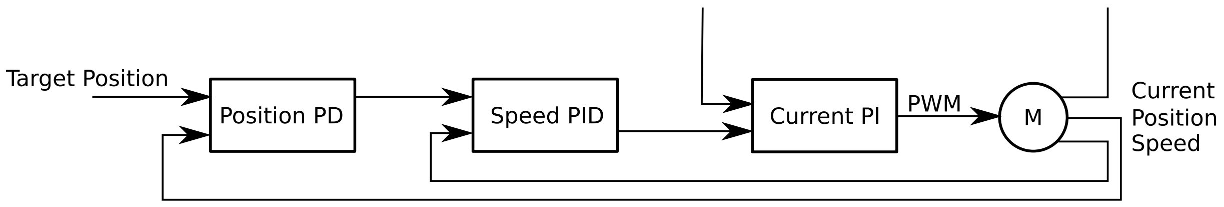


Figure 6.1: Standard Motor Module PID architecture.

Principle of Operation

The controller of the motor module is a triple PID controller nesting current, speed and position sub-controllers. Thus there are one PID for setting the current to the motor, one to set the speed, and one to set the position, as shown in Figure 6.1.

This architecture is flexible as it allows to set limits on maximum current or maximum speed. One can disable the position controller and directly feed the input of the speed controller to directly control the motor speed. One can apply the same override to the current controller. The following pseudocode shows the generic code of the PID controller. The PD or PI controllers are architecturally similar with the corresponding term set to zero.

$$i(t) = i(t-1) + e(t) \tag{6.1}$$

$$\text{output} = \frac{e(t) \cdot \text{Kp} + i(t) \cdot \text{Ki} + \text{Kd} \cdot (e(t) - e(t-1))}{\text{scaler}} \tag{6.2}$$

Moreover the speed and position time constants are not in the same order of magnitude than the current ones. Thus the current controller must run at a higher frequency than the speed and position controllers.

The current controller filters the current through an IIR filter with the same time constant as the motor thermal time constant. This ensures that the current imposed to the motor is never higher than its nominal current.

Variables

`M.pid.current_max`
: The absolute value of the maximum current in the motor in milliampere. If higher than the nominal current the controller will internally automatically reduce it to avoid overheating. This allows to set the maximum torque of the motor.

`M.pid.speed_max`
: The maximum speed of the motor. The unit depends on implementation.

  Warning: The maximum speed is not enforced if the controller is set in current mode.

`M.pid.enable`
: The selection of the controller:

  0 The controller is disabled, the motor is short-circuited.

  1 Only the current controller is enabled. Use `M.pid.target_current` to set the target current.

  2 The speed control is enabled (thus the speed and current controllers are enabled). Use `M.pid.target_speed` to set the target speed.

  3 The position control is enabled (thus the speed, current and position controllers are enables). Use `M.pid.target_position` to set the target position.

`M.current`
: The actual current in the motor, in milliampere. Only updated if the PID controller is enabled.

`M.speed`
: The actual speed of the motor. The unit depends on implementation. Only updated if the PID controller is enabled.

`M.position`
: The actual position of the motor. The unit depends on implementation. Only updated if the PID controller is enabled.

`M.pid.target_current`
: The target current of the current controller, in milliampere.

`M.pid.target_speed`
: The target speed of the speed controller. The unit depends on implementation.

`M.pid.target_position`
: The target position of the speed controller. The unit depends on implementation.

Hidden Variables

You should not need to read nor to write these varibles in the basic Standard Motor Module use case.

`M.pid._period`
: The period, in millisecond, of the current controller. If positive and `M.pid.enable` is 1, 2 or 3 the PID is executing. If negative the PID is not executing and `M.pwm` is directly applied without any safeguard on the motor. The `M.pid.timer` event is generated at the specified period (absolute value). The maximum period is 400 ms.

`M.pid._kp_i`
: The Kp term of the current controller, must be positive.

`M.pid._ki_i`
: The Ki term of the current controller, must be positive.

`M.pid._scaler_i`
: The scaler term of the current controller, must be ≥ 0.

`M.pwm`
: The PWM applied to the motor.

`M.pid._prescaler_s`
: The period multiplier of the speed and position controller. Must be > 0.

`M.pid._current_speed_pulse`
: The actual speed of the motor in raw units.

`M.pid._target_speed_pulse`
: The target speed of the motor in raw units.

`M.pid._kp_s`
: The Kp term of the speed controller, must be positive.

`M.pid._ki_s`
: The Ki term of the speed controller, must be positive.

`M.pid._kd_s`
: The Kd term of the speed controller, must be positive.

`M.pid._scaler_s`
: The scaler term of the speed controller, must be ≥ 0.

`M.pid._speed_max_pulse`
: The maximum speed in raw units. This value is automatically overwritten each time `M.pid.speed_max` is changed.

`M.pid._kp_p`
: The Kp term of the position controller, must be positive.

`M.pid._kd_p`
: The Kd term of the position controller, must be positive.

`M.pid._scaler_p`
: The scaler term of the position controller, must be ≥ 0.

`M.pid._target_position_pulse`
: The target position in raw units.

`M.pid._nominal_current`
: The nominal current of the motor, in milliampere.

`M.pid._time_cst`
: The thermal time constant of the motor, in millisecond.

`M.pid._override`
: Set to one allow the motor to exceed the software-defined bounds.

`M.enc._pulse`
: The motor's position in raw units.

`M._raw_current`
: The motor's current in raw units.

Events

`M.pid.timer`
: This event is triggered at the end of execution of every speed controller.

  Note: Even if the motor controller is set to current mode, the event will be triggered at the rate of the speed controller.

`M.overcurrent`
: This event is triggered when the mean current of the motor is higher than its nominal current. This implies that the maximum current of the motor is reduced to the nominal current of the motor.

`M.overcurrent.cleared`
: This event is triggered after `M.overcurrent` when the mean current of the motor is back lower than 90% of the nominal current.

Natives Functions

There is no native function for the Standard Motor Module.

Settings

You should not alter the settings, unless you know what you are doing. All the position are relative to the start of the Standard Motor Module section.

0 Kp of the current controller.

1 Ki of the current controller.

2 Scaler of the current controller.

3 Prescaler of the speed and position period.

4 Kp of the speed controller.

5 Kd of the speed controller.

6 Ki of the speed controller.

7 Scaler of the speed controller.

8 Maximum current allowed in the motor, in milliampere.

9 Nominal current of the motor, in milliampere.

10 Motor's heating time constant, in millisecond.

11 Maximum speed allowed, unit is implementation dependant.

12 Kp of the position controller.

13 Kd of the position controller.

14 Scaler of the position controller.

15 Period of the current controller, in millisecond.

16 Raw offset of the current.

17 Software maximum bound on position.

18 Software minimum bound on position. If the maximum and minimum positions are equals, the software end stops are disabled.

6.2.4 Treel-left

This aseba node manage the following sensors and actuators:

- Left treel[2].
- Two left-side, close range, infrared ground sensors.
- Three axis accelerometer.
- Three axis gyroscope.
- 15.56Mhz ISO 15693 RFID host.

Due to some internal limitation the two infrared ground sensors, the gyroscope and the accelerometer are bound to the same sampling speed.

This module embedd a standard motor module named `motor`. The speed and distance unit is the motor "pulse". For a more convenient unit, a special virtual encoder is available which unit is the thenth milimeter. Please refer to section 6.2.3 for detailed documentation of this module.

Variables

`motor.enc.period`
The period in ms of the virtual encoder. Maximum is 400ms. 0 mean disabled.

`motor.enc.delta_pulse`
The number of motor pulse (raw unit) since the last virtual encoder event.

`motor.enc.delta_dist`
The traveled distance in $\frac{1}{10}$ mm since the last virtual encoder event.

`sensors.period`
The update period of the ground and imu sensors. Maximum is 400ms, minimum is 10ms. 0 mean disabled.

`ground.corrected`
The calibration-corrected value of the two ground sensors. The calibration should produce a value of 1000 when the robot is on a white paper sheet.

`ground.ambiant`
The ambiant infrared value. This value is read just before switching on the infrared LED of the sensor.

`ground.reflected`
The reflected infrared value. This value is read 300us after the infrared LED of the sensor has been switched on.

[2] Wheel and track

`ground.delta`
: This is strictly equal to `ground.reflected` minus `ground.ambiant`.

`imu.enable`
: 0 mean accelerometer and gyroscope are powered off. Write 1 to power them on.

`imu.acc.corrected`
: The three accelerometer axis in X, Y and Z order without offset. The unit is ≈ 993 code/g. The range is $\pm 1.5g$

`imu.acc.raw`
: The three accelerometer axis in X, Y and Z order without offset cancelation. The unit is the same as `imu.acc.corrected`.

`imu.gyro`
: The three gyroscrope axis. Thoses value have an offset which change with temperature. The drift become noticeable after about 2-5 minutes. The unit is ≈ 31.2 code/$\frac{\circ}{s}$. The manufacturer of the gyroscope give a -50%, +100% tolerance on the scale factor.

`rfid.enable`
: 0 mean the rfid chip is powered down. Write 1 to power it up. You must power up the rfid chip before using any of the rfid native functions.

`timers.period`
: The period of four general purpose timers. 0 mean disabled. Maximum period is 400ms. The `timer`*`X`* event expire each time the period timeout.

`interaxis`
: The distance between the two wheel of the robot in $\frac{1}{10}$ mm.

Events

`motor.enc.timer`
: This event is trigged each time the virtual encoder timer expire. See the variables `motor.enc.*` for more informations.

`sensors.updated`
: This event is trigged each time new sensors values are available. See the variables `sensors.period`, `ground.*`, `imu.gyro` and `imu.acc.*`.

`timer`*`X`*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

`rfid.scan_done`
: This event is trigged when the scanning operation of the rfid chip is over. You can check the result of scan with the `rfid.count()` native function. See also the `rfid.scan()` native function.

`rfid.read_done`
: This event is trigged when the read operation on a rfid tag is done. The result of the scan is now available in the argument `data` passed to the `rfid.read()` native function.

`rfid.write_done`
: This event is trigged when the write operation on the rfid tag is done. The status (success/failure) is available in the `failed` argument passed to the `rfid.write()` native function.

Natives functions

`rfid.scan`
: Start a RFID scan. This will initiate a standard enumeration of all the RFID tags in close contact with the antenna. A maximum of five tag will be discovered, this should not be a problem as the antenna can power a maximum of 3 tags. This native function will trigger the `rfid.scan_done` event when the enumeration process is over.

`rfid.count(n)`
: Get the number of RFID tags found by the last enumeration.

`rfid.result(n, id, rssi, block, count)`
: Get the ID, received signal strength indication, block size and block count of the tag number N.

  `n` Tag number. Must be < to the number returned by `rfid.count()`.

  `id` The 64bits unique identifier of the tag.

  `rssi` Received signal strength indication. Minimum value: 0, maximum value: 31.

  `block` The tag block size in bytes.

  `count` The tag block count.

`rfid.read(i, b, d, failed)`
: Initiate a read operation of the EEPROM block *b* of the RFID tag with unique identifier *i*. When finished the `rfid.read_done` event is generated. If *failed* is equal to 0, the data read are placed in the *d* If *failed* is nonzero the read has failed, thus *d* is noninitialised.

  Note: The variables *failed* and *d* must not be touched between the call to `rfid.read()` and the `rfid.read_done` event.

`rfid.write(i, b, d, failed)`
: Initiate a write operation to the EEPROM block *b* of the RFID tag with unique identifier *i*. When finished the `rfid.write_done` event is generated. If *failed* is equal to 0, the data has been succefully written. If *failed* is nonzero the write operation has failed.

  Note: The variables *failed* and *d* must not be touched between the call to `rfid.write()` and the `rfid.write_done` event.

Settings

The settings number 0 to 18 are used by the standard motor module.

19, 20
: Ground sensors calibration offset.

21, 22
: Ground sensors calibration gain.

23 Distance calibration value. Number of motor pulse per straight 50cm displacement of the robot.

24 Rotation calibration value. Number of motor pulse per full trun over itself of the robot.

25 . . . **27**
: Accelerometer offsets.

6.2.5 Treel-right

This aseba node manage the following sensors and actuators:

- right treel.
- Two right-side, close range, infrared ground sensors.
- Supercapacitor charge.

Due to some internal limitation the two infrared ground sensors and the capacitor charge voltage are bound to the same sampling speed.

This module embedd a standard motor module named `motor`. The speed and distance unit is the motor "pulse". For a more convenient unit, a special virtual encoder is available which unit is the thenth milimeter. Please refer to section 6.2.3 for detailed documentation of this module.

Variables

`motor.enc.period`
: The period in ms of the virtual encoder. Maximum is 400ms. 0 mean disabled.

`motor.enc.delta_pulse`
: The number of motor pulse (raw unit) since the last virtual encoder event.

`motor.enc.delta_dist`
: The traveled distance in $\frac{1}{10}$ mm since the last virtual encoder event.

`ground.period`
: The update period of the ground sensors and the capacitor charge. Maximum is 400ms, minimum is 10ms. 0 mean disabled.

`ground.corrected`
: The calibration-corrected value of the two ground sensors. The calibration should produce a value of 1000 when the robot is on a white paper sheet.

`ground.ambiant`
: The ambiant infrared value. This value is read just before switching on the infrared LED of the sensor.

`ground.reflected`
: The reflected infrared value. This value is read 300us after the infrared LED of the sensor has been switched on.

`ground.delta`
: This is strictly equal to `ground.reflected` minus `ground.ambiant`.

`battery.capacitor`
: The charge voltage of the supercapacitors. The unit is the mV.

`timers.period`
: The period of six general purpose timers. 0 mean disabled. Maximum period is 400ms. The `timer`*X* event expire each time the period timeout.

`interaxis`
: The distance between the two wheel of the robot in $\frac{1}{10}$ mm.

Events

`motor.enc.timer`
: This event is trigged each time the virtual encoder timer expire. See the variables `motor.enc.*` for more informations.

`ground_sensors.updated`
: This event is trigged each time new sensors values are available. See the variables `ground.*` and `battery.capacitor`.

`timer`*X*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

Natives functions

There is no native functions specific to this virtual machine.

Settings

The settings number 0 to 18 are used by the standard motor module.

19, 20
: Ground sensors calibration offset.

21, 22
: Ground sensors calibration gain.

23 Distance calibration value. Number of motor pulse per straight 50cm displacement of the robot.

24 Rotation calibration value. Number of motor pulse per full trun over itself of the robot.

6.2.6 Sensors

This aseba node manage the following sensors:

- 24 horizontal infrared proximity bumpers.
- 8 vertical infrared sensors.

As the differents proximity sensors must be synchronized to ensure they don't interfer with each others, they are bound to the same sampling rate.

Variables

`proximity.ambiant`
: The raw ambiant infrared value of the horizontal sensors. The sensor number is counted clockwise starting from the front of the robot. This value is read just before switching on the infrared LED of the sensor.

`proximity.reflected`
: The raw reflected infrared value of the horizontal sensors.

`proximity.delta`
: This is equal to `proximity.reflected` minus `proximity.ambiant`.

`proximity.corrected`
: This is the processed sensor value. All the sensors should have a near zero offset when no object is present and a somewhat identical gain.

`ground.ambiant`
: The raw ambiant infrared value of the vertical sensors.

`ground.reflected`
: The raw reflected infrared value of the vertical sensors.

`ground.delta`
: This is strictly equal to `ground.reflected` minus `ground.ambiant`.

`ground.corrected`
: This is the processed sensor value. All the sensors should have a near zero offset when no object is present and a somewhat identical gain.

`sensors.period`
: The sampling period of the sensors in ms. The maximum value is 400ms, the minimum is 5ms. 0 mean the sensors are disabled.

`sensors.bitfield`
: This is a two word (32bits) bitfield where a 1 indicate the sensor is switched on and 0 mean the sensor will not be switched on during the scanning process. The bits number 0 to 23 are used for the horizontal sensors and the bits 24 to 31 are used for the vertical sensors.

`timers.period`
: The period of six general purpose timers. 0 mean disabled. Maximum period is 400ms. The `timer`*`X`* event expire each time the period timeout.

Events

`sensors.updated`
: This event is trigged each time new sensors values are available.

`timer`*X*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

Natives functions

There is no native functions specific to this virtual machine.

Settings

0 . . . **23**
: Horizontal sensors calibration offset.

24 . . . **47**
: Horizontal sensors calibration gain.

48 Horizontal sensors clamp value.

49 . . . **56**
: Vertical sensors calibration offset.

57 . . . **64**
: Vertical sensors calibration gain.

6.2.7 sensor-turret

This aseba node manage the following sensors and actuators:

- Two long range infrared sensors (20cm to 1.5m).
- Two short range infrared sensors (0cm to 30cm).
- Rotating motor.

The sampling rate of the sensors is fixed to 60Hz.

Variables

`sharp.value`
: The raw value from the sensors.

`sharp.dist`
: The computed distance in mm.

`sharp.angle`
: The position of the sensors in degree.

`voltage`
: The voltage of the secondary controller.

Events

`sharp.updated`
: New sensors values are available.

`undervoltage`
: The secondary controller experienced an undervoltage condition. The sensors value can be degraded.

`sharp.disconnect`
: The secondary controller had a communication or power failure, thus disconnected the sensors. You can restart it with `sharp.start()`.

Natives functions

`sharp.start`
: Start to power the sensors. You can call the `sharp.set_speed`, `sharp.set_position` or `sharp.send_conf` before powering up the sensor.

`sharp.stop`
: Power down the sensors. Call this when you don't need the sensors anymore as it consume quite a large amount of power (about 2W).

`sharp.set_speed(s)`
: Set the sensors' rotating speed s in RPM.

`sharp.set_position(p)`
: Set the sensors' position p in degree.

`sharp.send_conf`
: Force the configuration transfert to the sensors. The natives functions `sharp.set_speed` and `sharp.set_position` already schedule a such transfert.

Settings

0 . . . 10
: X calibration value for sensor 0

11 . . . 21
: Y calibration value for sensor 0

22 . . . 32
: X calibration value for sensor 1

33 . . . 43
: Y calibration value for sensor 1

44 . . . 54
: X calibration value for sensor 2

55 . . . 65
: Y calibration value for sensor 2

66 . . . 76
: X calibration value for sensor 3

77 . . . 87
: Y calibration value for sensor 3

Hidden Variables

`motor._command`
: The raw value to send to the motor. If position control is wanted the command should be computed as:

$$-position - 1$$

. The position must be given in raw unit. If speed control is required the command should be computed as

$$speed + 16384$$

. The speed must be given in raw unit.

`sharp._led`
: The led used for the scan for each sensor. Each item in the array is a 8 bits number with the first nibble correspondig to the first led to use and the second to the second led. The led are numeroted from 1 to 5. Use 0 to switch off one led. The default value is 3.

`sharp._raw_position`
: The position of the sensor in raw unit.

6.2.8 Led-rgb

This aseba node manage the following sensors and actuators:

- One RGB 3W beacon LED.
- One LED's temperature sensor.

The temperature sensor is thermally coupled to the led. If the temperature is above 80°C, the led will switch off automatically and emit the `led.overheat` event. If will return to normal operating state when the temperature decrease below 70°C and emit the `led.overhead.cleared`.

Variables

`led.temperature`
: The led temperature in 0.1 °C unit.

`led.temperature.period`
: The led temperature refresh period in ms. Maximum value is 400ms. This trigger the `led.temperature.updated` event.

`timers.period`
: The period of seven general purpose timers. 0 mean disabled. Maximum period is 400ms. The `timer`*X* event expire each time the period timeout.

Events

`led.overheat`
: The LED has overheated, it is temporarly switched off, it will be automatically switched on when the temperature is sufficiently lower.

`led.overheat.cleared`
: The LED overheat condition is cleared, if the LED was on it is automatically switched on.

`led.temperature.updated`
: The LED temperature is updated. You can read the new temperature in the `led.temperature` variable.

`timer`*X*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

Natives Functions

`led.rgb(r,g,b)`
: Set the LED RGB brightness for each red (`r`), green (`g`) and blue (`b`) componnant. The maximum value for each componnant (full brighness) is 1000. The minimum value (switched off) is 0.

Settings

There is no settings with this module.

Hidden Variables

There is no specific hidden variables in this module.

6.2.9 gripper-sensor

This aseba node manage the force sensor of the marXbot. The general units are in metric gram.

Variables

`force.period`
: The update period of the force sensor, in ms. Max value is 400ms.

`force.mag`
: The magnitude of the force sensor in metric gram.

`force.angle.aseba`
: The angle of the force vector. The unit is aseba angle see section 6.2.1.

`force.angle.deg`
: The angle of the force vector, in degree [0-360].

`force.x`
: The force on the X axis, in metric gram.

`force.y`
: The force on the Y axis, in metric gram.

`timers.period`
: The period of six general purpose timers.

Events

`sensor`
: This even is trigged each time a new force mesurement is available.

`timer`*X*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

Natives Functions

There is no natives function specific to this module.

Settings

0	X offset
1	Y offset

Hidden Variables

`force._raw_x`
: The raw X axis force mesurement without offset cancelation.

`force._raw_y`
: The raw Y axis force mesurement without offset cancelation.

`_potx.tot_r`
: The X axis potentiometer end-to-end total resistance, in 0.1 kOhm.

`_potx.wpos`
: The wipers positions of the X axis potentiometers.

`_poty.tot_r`
: The Y axis potentiometer end-to-end total resistance, in 0.1 kOhm.

`_poty.wpos`
: The wipers positions of the Y axis potentiometers.

Hidden Natives Functions

`_potx.set(pot, value)`
: Set the X axis potentiometer *pot* wiper position to *value*.

`_potx.update()`
: Read back the X axis potentiometers wiper position.

`_potx.save(pot)`
: Save X axis potentiometer *pot* wipers position in EEPROM.

`_poty.set(pot, value)`
: Set the Y axis potentiometer *pot* wiper position to *value*.

`_poty.update()`
: Read back the Y axis potentiometers wiper position.

`_poty.save(pot, value)`
: Save Y axis potentiometer *pot* wipers position in EEPROM.

6.2.10 griper-led

This node manage the gripper and led ring of the marXbot. Is manage two motors, one for the gripper fingers, and the other to manage the gripper rotation. The finger motor use a standard motor module interface named `gipper`. The speed unit is degree/100ms and the position is in degree. The rotation motor use a standard motor module interface named `rev`. The speed unit is degree/10ms and the position is in degree. Please refer to section 6.2.3 for detailed documentation of both motors.

Variables

`timers.period`
: The period of two general purpose timers.

Events

`timer`*X*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

Natives Functions

`set_led(n,r,g,b)`
: Set the LED RGB brightness for each red (`r`), green (`g`) and blue (`b`) componnant for the led number `n`. The maximum value for each componnant (full brighness) is 63. The minimum value (switched off) is 0. The led number 0 is the first led. The led number 11 is the last led.

`gripper.enc.reset`
: Reset the gripper motor position. Usefull to set the motor initial (0) position.

`rev.enc.reset`
: Reset the rotation motor position. Usefull to set the motor initial (0) position.

Settings

There is no others settings than the standard motor modules settings.

6.2.11 rab2

This section describe the Range and Bearing module of the marXbot. The Range and Bearing use the infrared medium to sens the relative position of other surrounding robots and 2.4Ghz radio to synchronize the swarm and exchange data. The RF and infrared medium access are both TDMA and syncronized using an unique address on each node and a predetermined network size. This implies the following restrictions:

- Each node of the network must have an unique ID.
- Each node of the network must use the same network size.
- Each node of the network must use the same medium access time.

Variables

`rf.id`
: The ID of the RAB on the network. Must be unique on the network

`rf.swarm_size`
: The swarm size. Must be higher than 3 and smaller than 253.

`rf.slot_time`
: The medium access time in microsecond. Minimal value: 3000us. Recommanded value: 4000us.

`rf.max_tx`
: The number of time the RAB module will send a packet without having any answer. Put 0 to disable this behavior.

`rf.tx_data`
: The data to send. The first 13 bytes will be transmitted (little endian).

`rf.ev.node_id`
: The node concerned by the event. See events `node.new`, `node.lost`.

`tx.power.en`
: Enable the TX power amplifiers. Only impact infrared medium. If TX is disabled the node will be visible on the RF network, but not on the infrared.

`rx.power.en`
: Enable the RX power amplifiers. Only impact infrared medium. If RX is disabeld the node will not be able to sens the others robot position but will be able to communicate over the RF.

`rx.source`
: The received's ID.

`rx.data`
: The data payload of the received packet. Only the first 13 bytes are valid (little-endian).

`rx.range1`
: The first stage amplifiers value on the IR sensors.

`rx.range2`
: The second stage amplifiers value on the IR sensors.

`rx.range3`
: The third stage amplifiers value on the IR sensors.

`rx.dist`
: The estimated distance of the emiting robot. Units are mm.

`rx.angle`
: The estimated angle of the emiting robot. Units are aseba angle, see section 6.2.1.

`timers.period`
: The period of four general purpose timers.

Events

`node.new`
: This event is emitted when a previously never seen node is emiting a packet. See `rf.ev.node_id`.

`network.down`
: This event is emitted when the RAB stop to emit because no answer has been received for a too long time. See `rf.max_tx`.

`network.start`
: This event is emitted when the RAB is creating a new network.

`node.lost`
: This event is emitted when a node which was previously emiting stopped emiting for more than 255 packets.

`packet.rx`
: This event is emitted when a new packet is received on the RF.

`packet.tx`
: This event is emitted when a new packet is transmitted on the RF.

`timer`*X*
: Thoses event are trigged each time the corresponding timer expire. See the variables `timers.period`.

Natives Functions

`rf.start()`
: Start the RF network.

`rf.stop()`
: Stop the RF network.

Settings

0 ... **7**
: Horizontal IR sensors first stage gain saturation values

8 ... **15**
: Horizontal IR sensors second stage gain saturation values

16 ... **23**
: Horizontal IR sensors third stage gain saturation values

24 ... **71**
: Calibration values

72 The default RAB ID

73 The default network size

74 The default slot time

75 The RF frequency to use

Hidden Variables

`rx._dist_mm`
: The estimated source distance on each horizontal IR sensors in millimeters.

6.2.12 Magnetic-Gripper-Main

This section describes the Magnetic-Gripper-Main node of the Marxbot. This node lies in the base of the general module called Magnetic-Gripper. It is bound with the Magnetic-Gripper-Front node which contains the gripper, some sensors and the RFID tag system.

In the Magnetic-Gripper-Main node, you will find three motors called *mot_lift*, *mot_rot* and *mot_tilt* which are responsible respectively of the gripper lifting, the turret rotation and the gripper tilting around its central axis. The motors are controlled through the standard motor module, see 6.2.3.

Beside the motors, this node has one strain gauge, measuring the lateral bending of the arms.

Variables

Most of this node's variables concern the motors. For a description of each of these variable, see 6.2.3.

`gauge_bend.value`
: This is the value read by the strain gauge measuring the lateral bending of the gripper's arms. This value can give a result between 0 and 4095. If correctly calibrated, it should be around 2048 when the arms are not bent. When the variable displays a value bigger or smaller than 2048 by 1500, it means that a force of approximately 5000 N is applied laterally to the gripper. The calibration method used a mass of 500 gramms attached to the gripper to determine the gain factors. This means that if `gauge_bend.value` displays 3548, an force equivalent to a dead mass of 500 gramms is pressing laterally on the gripper in one direction and if `gauge_bend.value` displays 548, the same force is applied in the other direction.

`gauge.period`
: This is the period between two measurement of the strain gauges. It can be set between 1 and 1000. The smaller the period is, the busier the microcontroller will be. A good value is typically 100 (meaning the sensor is updated every 0.1 second).

`led[0], led[1]`
: These leds are not visible unless you demount the robot. They are used in debugging phases.

`timers.period[0...3]`
: These are three timers that you can set to a different period. The maximum period is 400 ms.

Events

Most of the events concern the motors, for more details see 6.2.3.

`mot_rot_index`
: This event is emitted when the rotation motor reach the index. This happen when the robot is aligned with its treels, by default it is the position 0 of this motor.

`gauge.updated`
: This event is emitted when the strain gauges are updated.

`timer0, timer1, timer2`
: These events are emitted when the timer 0, 1 or 2 gives a clock count.

Natives Functions

The only three native functions of this node concern the reset of the motors encoder. You can use them with: call mot_lift.enc.reset(), call mot_rot.enc.reset() or call mot_tilt.enc.reset(). These functions will set the encoders of the motors to 0.

If you change the reference of the motors, be careful not to go out of the borders the motors can reach. This could destroy the gripper-arms part.

6.2.13 Magnetic-Gripper-Front

This section describes the Magnetic-Gripper-Front node of the Marxbot. This node lies in the gripper of the module called Magnetic-Gripper. It is bound with the Magnetic-Gripper-Main node.

In the Magnetic-Gripper-Front node, you will find 22 IR sensors, two leds (not visible if the robot is mounted) and an RFID scanner.

The IR sensors are splitted into four groups. The first ones, IR[0. . . 9], are the frontal sensors. The second ones, IR[10. . . 19], are the bottom ones. The last two are the left and right sensor, IR[20] and IR[21]. The values of these sensors vary from 0 (nothing is in front of the sensor) and 4095 (something is sticked to the sensor).

Variables

`ir.period`
: This is the period between two measurements of the IR sensors. If set to 0, the sensors are deactivated. If you want to use the microphones, the variable `ir.period` cannot be set to zero (otherwise, the microphone are not powered).

`ir.ambiant`
: These 22 values give the ambiant value measured by the IR sensors.

`ir.reflected`
: These 22 values give the reflected value measured by the IR sensors.

`ir.delta`
: These 22 values give the difference between the ambiant and the reflected value of the IR sensors. These are the value to be used to measure a distance. This will not take account of the ambiant lightning (as long as the sensor is not saturated).

`led[0], led[1]`
: These leds are not visible unless you unmount the robot. They are used in debugging phases.

`rfid.enable`
: This is a bool value enabling the rfid module (1) or disabling it (0).

`timers.period[0...2]`
: These are two timers that you can set to a different period. The maximum period is 400 ms.

Events

`ir` This event is emitted when the IR sensors are updated.

`rfid.scan_done`
: This event is trigged when the scanning operation of the rfid chip is over. You can check the result of scan with the `rfid.count()` native function. See also the `rfid.scan()` native function.

`rfid.read_done`
: This event is trigged when the read operation on a rfid tag is done. The result of the scan is now available in the argument `data` passed to the `rfid.read()` native function.

`rfid.write_done`
: This event is trigged when the write operation on the rfid tag is done. The status (success/failure) is available in the `failed` argument passed to the `rfid.write()` native function.

`timer0, timer1`
: These events are emitted when the timer 0 or 1 gives a clock count.

`sound`
: This event is emitted when the sound buffer of both microphones is filled.

Natives Functions

`rfid.scan`
: Start a RFID scan. This will initiate a standard enumeration of all the RFID tags in close contact with the antenna. A maximum of five tag will be discovered, this should not be a problem as the antenna can power a maximum of 3 tags. This native function will trigger the `rfid.scan_done` event when the enumeration process is over.

`rfid.count(n)`
: Get the number of RFID tags found by the last enumeration.

`rfid.result(n, id, rssi, block, count)`
: Get the ID, received signal strength indication, block size and block count of the tag number N.

  `n` Tag number. Must be < to the number returned by `rfid.count()`.

  `id` The 64bits unique identifier of the tag.

  `rssi` Received signal strength indication. Minimum value: 0, maximum value: 31.

  `block` The tag block size in bytes.

  `count` The tag block count.

`rfid.read(i, b, d, failed)`
: Initiate a read operation of the EEPROM block *b* of the RFID tag with unique identifier *i*. When finished the `rfid.read_done` event is generated. If *failed* is equal to 0, the data read are placed in the *d* If *failed* is nonzero the read has failed, thus *d* is noninitialised.

  Note: The variables *failed* and *d* must not be touched between the call to `rfid.read()` and the `rfid.read_done` event.

`rfid.write(i, b, d, failed)`
: Initiate a write operation to the EEPROM block *b* of the RFID tag with unique identifier *i*. When finished the `rfid.write_done` event is generated. If *failed* is equal to 0, the data has been succefully written. If *failed* is nonzero the write operation has failed.

  Note: The variables *failed* and *d* must not be touched between the call to `rfid.write()` and the `rfid.write_done` event.

`sound.buffer(l[128],r[128])`
: This function gets the two buffer of the microphones. The left and right buffer is a table of 128 elements. If you take the difference between the maximum and the minimum value of each table, you will get the volume of the recorded sound.

`servo.open()`
: This function disables the magnetic gripper.

`servo.close()`
: This function enables the magnetic gripper.

6.2.14 Magnetic-Gripper Motors calibration

This section will describe the motors calibration process. Indeed, the tilting and lifting motors must not go beyond some physical limits illustrated on 6.2. As long as no absolute landmark exists on which these motors can rely, a calibration process has been established.

The Initial_calibration.aesl file will perform this calibration.

1. The robot will turn on itself using the rotation motor. This is the only motor with an absolute index. As soon as it has reached this index, the robot knows it is in the right position. It will then turn until it reaches the index and reset the rotation motor encoder to set the zero position in parallel with the treels.
2. The robot will lower the gripper until it raise itself a bit, then will set its arms a bit higher than the ground. This manoeuvre is done to detect approximately the ground.

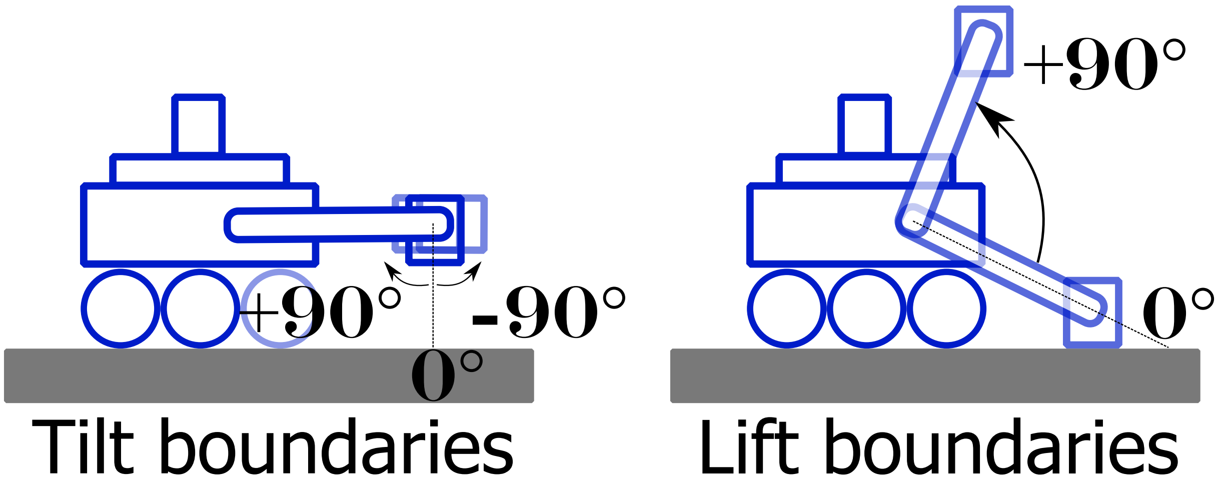


Figure 6.2: The boundaries of the lifting and tilting motors.

3. The robot will tilt its gripper around itself (beyond the boundaries) to detect the ground and to be able to set the tilting motor to its zero. Once done, the gripper returns to its new zero position.
4. The lifting arms will go down again to adjust the ground detection. When it detected the ground, this time through the IR sensors, the robot is stopped and the three motors are calibrated to their respective 0.

The robot should lie on a flat, obstacle-less, white surface during the calibration. If during the phase 2 the robot doesn't stop lifting itself after a few second, you should press on the dsPic Reset to stop the calibration process, set the arms to a position included in the boundaries and restart the calibration process. The lifting and tilting motors must be in a position included in the boundaries before initiating the calibration process. If at the end of the phase 3 the gripper is not at its 0 position, the calibration process must be stopped (dsPic Reset) and restarted.

This calibration process can be ineffective on a few robots (it depends on the IR values and the motor current feedback which can change from one robot to another). If you prefer, you can place the robot at the desired position yourself and the launch the three functions call mot_lift.enc.reset(), call mot_rot.enc.reset() and call mot_tilt.enc.reset() which will set the motors encoders to 0 (the "zero" position is when the turret is aligned with the treels and when the gripper lies on the floor with the magnetic gripper at 0° as on 6.2). The three motors are reversible, which means that you can move them by hand. But be careful, sometimes, the mechanical links between the motor and the part you want to move can be a bit stiff. The easiest method to place the motor where you want by hand is to set the variables `mot_lift.pid.enable`, `mot_tilt.pid.enable` and `mot_rot.pid.enable` to 1 and then to set `mot_lift.pid.target_current`, `mot_tilt.pid.target_current` and `mot_rot.pid.target_current` to 0. This means that the motors aimed current is 0. When you will try to move one of these motors, it will generate a current. Then, the motor will try to cancel this induced current by powering the motor in the direction you are pushing. In the end, the motor will help you move itself and the possible stiffness will disappear.

Bibliography

[1] M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada. The marXbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 4187–4193. IEEE.

[2] Michael Bonani. *Robotique collective et auto-assemblage.* PhD thesis, Lausanne, 2010.

[3] Stéphane Magnenat. *Software integration in mobile robotics, a science to scale up machine intelligence.* PhD thesis, Lausanne, 2010.